Beautiful *but* Boring

Writing Python Code for Future You

Beautiful but Boring

Writing Python Code for Future You

Roth Earl

Contents

To my wife — the rubric I live by, and the compass that always points true.

From the Author

This book grew out of a personal project.

Pyr-CLI is a toolkit of composable command-line programs I have been building for several years — not for work, not for anyone else, purely for the joy of it. If you have been programming long enough, you know what that feels like: a project that has no deadline, no stakeholders, no justification required. Something you return to because the work itself is satisfying. Pyr-CLI was that project. It is not a large codebase, but it is the kind of codebase that accumulates decisions about naming and types, about how functions should be structured and what they should promise. At some point I started asking language models to evaluate those decisions. Not to write code. To evaluate it.

It started as a list of about ten bullet points — things I wanted the evaluator to look for. Each one informal, somewhat vague — exactly the kind of thing you would write before you knew how much precision mattered. Even so, the evaluations were useful. Code that had seemed fine turned out to be harder to read than it was to write. Names that had seemed clear turned out to be describing mechanisms rather than contracts. Parameters that had seemed obvious turned out to be ambiguous at the call site.

So I refined the criteria. And refined them again. And again. There is a definition of insanity attributed, probably incorrectly, to Einstein: doing the same thing over and over and expecting a different result. By that definition, what I was doing qualified. Evaluate. Find something. Fix it. Evaluate again. Find something new. Fix that too. If you have ever been consumed by a project that mattered to you — the kind where you look up and realize the sun is coming up — you know the feeling. By the time the list had grown into a twelve-page Word document, I finally had a name for what it was: the *Python Code Evaluation Rubric*. I had not known, when I wrote the first bullet point, that I was writing a rubric. Each rule had a rationale, earned from a real finding. That iteration may have qualified as insanity. The rubric did not.

When I stopped pasting prompts into a browser window and started using Claude Code, something changed. The rubric lived in the repository. Evaluations ran as structured skills rather than manual copy-paste sessions. I could evaluate the same code three times in a single session, apply findings, and evaluate again. Each pass produced fewer findings and code that was more obviously right.

That iteration is what Part VII is about. Not AI tools — those are an implementation detail. The developer leads. Standards live in the rubric. AI evaluates against it; code quality is a property of a process, not a snapshot. A single evaluation is useful. An evaluation that closes the loop — where the developer reviews findings, accepts or rejects them, and updates the rubric before the session ends — is an investment. Each evaluation starts from a higher floor.

Over enough iterations, the code stopped producing findings. Not because the rubric became lenient. Because the code became honest. Names that kept their contracts. Types that stated exactly what the function required. Documentation that described observable behavior and nothing else. The rubric did not produce that codebase. Iteration did.

Chapter 28 is an annotated reading of that codebase. Not as an advertisement for the tools I used to get there, but as evidence that the principles in this book compound. Applied once, they improve a function. Applied consistently, across a codebase, over time, they produce something qualitatively different. Code that a reader encounters and immediately trusts. Every decision made deliberately. Every standard enforced enough times to have become habit. That is beautiful code. And because it is so utterly unsurprising, it is also boring. *Beautiful but Boring* is not a compromise. It is the goal.

That is what this book is trying to teach. The rubric-driven development chapters describe the mechanism. Everything else is the rationale for every rule in the rubric.

Start wherever the argument pulls you. But the chapters compound too.

Before You Begin

This book is for developers who already write Python and want to write it better. It is not a language tutorial — it does not explain what a function is, what a decorator does, or how to set up a virtual environment. The book takes for granted that you know the language, and asks a harder question: are you writing it well?

Recommended Reader Profile

The ideal reader takes their craft seriously — writing Python for a living, or simply for the love of it. They have shipped code that other people read. At some point they returned to something written months earlier and felt the slight disorientation of having lost the context behind it. They may not have thought carefully about why some code is harder to maintain than other code — but they have felt the difference.

Parts I through VI make the technical and philosophical case for boring code, chapter by chapter. Part VII connects those principles to the system that maintains them over time — the rubric-driven development framework. It is framed partly in terms of AI-assisted workflows and team consistency, but the underlying mechanism applies equally to a solo developer working without AI tools: a written standard that outlasts any single session, consistently applied, produces compounding improvements regardless of how the standard is enforced.

A reader who has maintained a non-trivial codebase will find every chapter immediately grounded. Those who have only written solo code will understand the mechanics but may not yet feel the motivation. The book becomes more actionable over time, as the problems it describes begin to feel familiar.

What You Must Already Know

These are the constructs the book draws on, not a pre-screening test. If you can follow a Python code review, you have enough to begin. The book explains the less common features — `match`, abstract collection types — when they appear.

The book does not teach these. It uses them.

Before starting:

- Variables, functions, loops, and conditionals
- Classes, `self`, and basic object-oriented design
- `try`/`except` with standard exception types
- How modules and imports work
- Standard library basics: `open`, `os.path`, common I/O patterns

Before Part II and beyond:

- Type hints: basic (`str`, `int`, `-> None`) and parameterized (`list[str]`, `dict[str, int]`)
- Keyword-only parameters — what $*$ does in a function signature
- Dataclasses: `@dataclass` and its basic options
- Custom exception classes
- `enum.Enum` and its members
- Comprehensions and generator expressions
- Decorator syntax — reading a decorated function, even if you have not written one

For full benefit in Chapters 13 and 16:

- Abstract collection types: `Iterable`, `Collection`, `Sequence`, `MutableSequence`
- The `match` statement and structural patterns (Python 3.10+)
- `typing.Protocol` and abstract base classes

Chapters 13 and 16 are the two points in the book where the assumed Python knowledge rises above intermediate. Both chapters explain the constructs they use, but a reader encountering `Iterable[T]` or a `match` pattern for the first time in those chapters will need to pause and learn the feature before the argument lands. Everything else stays at the intermediate level or below.

Would a Beginner Struggle?

An advanced beginner — someone who knows Python well but has not yet worked in a production or professional context — will face two barriers.

The first is mechanical. Chapters 13 and 16 assume familiarity with abstract collection types and `match` statement patterns. A beginner can follow most of the book without these, but those two chapters require a detour.

The second barrier runs through every chapter, and it cannot be bypassed by reading documentation. Most of the book's arguments derive their force from having felt the pain they describe. The rule "docstrings describe observable behavior, not implementation" is easy to memorize. It becomes urgent when you have debugged code whose docstring described an algorithm that was refactored away two versions ago. The book is readable by an advanced beginner. It is most valuable to someone who has already paid the cost of ignoring its principles.

If you have already maintained a non-trivial codebase — if the preface scenario landed as recognition rather than prediction — this book is immediately actionable. The principles describe problems you have already had.

If you have not yet maintained a non-trivial codebase — if you have not felt the compounding friction of poorly named functions, unclear signatures, or implementation-describing comments — read this book anyway. Some of it will feel abstract. Come back to those chapters when the examples stop feeling hypothetical.

A Note on Opinion

The rules in this book are not arbitrary preferences. Each one is grounded in PEP standards, widely adopted conventions, or patterns that have earned their place through production use. Where this book departs from convention — or takes a position on matters where reasonable developers disagree — it argues the case rather than asserting the conclusion. If a rule does not convince you, find the rationale and argue with that.

Preface

Somewhere along the way, you have opened a file, read a function name, and had no idea what it did. The name was short. The logic was dense. Comments — if there were any — described what the code did rather than why. You spent twenty minutes reconstructing the intent of something that should have taken thirty seconds to read.

The file was yours. You wrote it. Six months ago it was obvious.

This is not a story about forgetting. It is a story about code that was written for the wrong reader. The developer who wrote that function wrote it for themselves, in the moment, with full context. They knew what the data structure held, why the flag was named the way it was, and what the downstream caller would do with the result. The code made sense because everything that made it sensible was already in their head. None of it was in the file.

The developer who opened that file six months later — also you, with none of that context — was a different person entirely. That developer deserved better.

Every principle in this book is a response to that moment.

Not the dramatic version — the production incident, the 2am page, the cascading failure. Those happen too, and boring code helps there. But the daily version is what grinds developers down: the friction of reading code that should be transparent, the ten minutes spent decoding a variable name that could have been self-evident, the function that does two things but is named for one of them, and the type annotation that constrains the caller more than the function actually requires. None of these are catastrophes. Together, accumulated across a codebase and a career, they are exhausting.

Boring code is the antidote. Not simple code — boring code. There is a difference. Simple code solves a simple problem simply. Boring code is code that a reader encounters and

immediately understands, regardless of the problem's complexity. The names make true promises. The types state what the function actually needs. The documentation describes what callers can rely on. Structure makes the logic visible before the reader processes a single line.

Boring code is hard to write. It requires more thought at the keyboard than the clever alternative, not less. A one-liner is faster to write than a named function; an abbreviated variable name takes fewer keystrokes; the broad type annotation is easier to reach for than the accurate abstract one. Boring code costs effort upfront and pays it back every time someone reads it. Clever code does the opposite.

If that cost is not yet visible to you — if your code still reads clearly when you return to it, if clever solutions still earn you recognition and feel like mastery — you are the reader this book most needs to reach. The cost compounds slowly, and it compounds faster as the codebase grows and the team expands. This book is not waiting until the 2am call to make its case. It is making the case before the bill arrives, while there is still time to change what you write.

Boring code is not modest code. It is not code that lacks ambition or expertise. It directs its ambition toward the reader rather than the writer.

This book is not for beginners. It assumes that you write Python seriously — for a living or for the pleasure of it — that you have maintained code you wrote a year ago, and that you have felt the specific frustration of not being able to trust your own past decisions. The earlier chapters revisit fundamentals — naming, signatures, types — but from a perspective that only makes sense once you have written enough code to have made the mistakes the chapters describe. If you are new to Python, read a tutorial first. Come back when the 2am story sounds familiar.

It is also not a style guide. Style guides tell you where to put your spaces. This book argues about what your names promise, what your types say to callers, and what your comments are for. These are not stylistic concerns — they are semantic ones. A poorly named function is not ugly — it is misleading. A docstring that describes the implementation rather than the contract is not informal. It is wrong. This book defends its positions rather than decreeing them — each rule has a rationale, and the rationale is what you should argue with, not the rule.

The discipline this book teaches is the discipline of writing code as communication — specif-

ically, communication with the developer who will read it next. That developer may be a colleague. They may be someone who has not yet joined your team, or you, six months from now, with none of today's context. Whoever they are, they deserve code that respects their time, keeps its promises, and does not require them to read the implementation to understand the contract.

The title is a provocation. Beautiful code is the goal most programming books chase — admired, shared, described as elegant, clean, expressive. There is nothing wrong with beautiful code, except that beauty is often in conflict with the property that actually matters: that the next person to read it will understand it immediately, use it correctly, and be able to modify it without breaking something they did not know they were relying on.

Boring code is beautiful in a way that does not announce itself. The highest compliment a reader can pay your code is not "this is impressive." It is: "this was exactly what I expected." That sentence describes code that made no false promises, required no decoding, and contained no surprises. It describes code that was written for the reader, not the writer.

When every name keeps its promise, every type states its actual requirement, and every docstring describes the contract rather than the mechanism, the code becomes honest. Not impressive — honest.

That is what this book is about. Write it boring. Make it last.

Prologue — The Code You Cannot Use

Here is a function. You need to call it.

```python
# The clever version
def process_user_data(data, flag=False):
    if isinstance(data, str):
        data = json.loads(data)
    if flag:
        return [User(d["id"], d["name"]) for d in data if "id" in d]
    else:
        return [User(d["id"], d["name"]) for d in data]
```

Before you write the call site, you have questions.

What does `flag=True` do? The name says nothing. You have to open the function and read it to find out that it filters records missing an `id` key. That is not what "flag" suggests. It suggests a mode switch — `html` output, verbose logging, something behavioral. Filtering is transformation logic. The name is not wrong in a way that raises an error. It is wrong in a way that costs time.

Is this safe if `data` is invalid JSON? The function silently converts strings, but there is no indication of what happens when the JSON is malformed. Reading the implementation is the only way to know whether to wrap the call in a `try/except`.

Does this parse, filter, or construct? It does all three. The name says none of them.

So you write the call site on a guess. You write:

```python
users = process_user_data(data, True)
```

You assume `flag=True` is the safe path — the one that validates, or sanitizes, or handles edge cases. That is a natural reading of a boolean flag named `flag`. You skip the error handling because nothing in the name or signature suggested it was necessary. Six months later, someone passes data with malformed records. The function raises a `KeyError` on

`d["name"]` because the filtering only checked for `"id"`. The assumption was wrong. The function never promised anything about `"name"`. You just could not tell that without reading it.

This is not a contrived failure. It is the ordinary cost of code that cannot speak for itself.

Now here is the boring version of the same logic.

```
# The boring version
def parse_user_records(data: str) -> list[dict]:
    """Parse JSON data into a list of user records.

    - Raises ``JSONDecodeError`` if the data is not valid JSON.
    """
    return json.loads(data)


def normalize_user_records(records: Iterable[dict]) -> list[dict]:
    """Return records that include a required 'id' field.

    - Preserves input order.
    """
    return [record for record in records if "id" in record]


def build_users(records: Iterable[dict]) -> list[User]:
    """Construct User objects from validated records.

    - Raises ``KeyError`` if a record is missing required fields.
    """
    return [User(record["id"], record["name"]) for record in records]
```

The call site writes itself.

```
records = parse_user_records(data)
valid_records = normalize_user_records(records)
users = build_users(valid_records)
```

You know `parse_user_records` raises on invalid JSON because it says so. You wrap it in a `try/except` before you open the function, because the name and the docstring together tell you everything you need. You know `normalize_user_records` filters — not validates, not transforms — because that is what normalize means and what the docstring confirms. You know `build_users` can raise a `KeyError` because the docstring says so, and you pass

it only the output of `normalize_user_records`, which has already removed incomplete records. The pipeline is safe. You assembled it from the names and signatures alone.

Nothing in the boring version is clever. None of it will earn a comment in a pull request. It is three functions where one existed before, each doing one thing, none hiding a surprise. A reader who encounters it processes it and moves on.

That is the whole argument.

This book is about the gap between those two versions — what causes it, what it costs, and how to close it. Not just in one function, but across an entire codebase, maintained over time, by developers who were not there when it was written. The principles that separate the boring version from the clever one have names and rationale. They are not instinct. They are a discipline.

The chapters that follow build that discipline, one decision at a time.

Introduction

This book is organized in eight parts, moving from philosophy to practice to the system that sustains both. The parts build on each other, but they are not a dependency chain. A developer who wants to go directly to naming conventions can start at Part II. The rubric-driven development framework in Part VII can be read independently of the chapters that precede it. The chapters are written to be navigable — each one stands on its own argument, with cross-references where an earlier principle is being extended rather than re-established.

The book rewards sequential reading. The decisions made in Part II about what names promise inform everything in Parts III and IV about what signatures and types communicate. Documentation disciplines in Part VI are extensions of the contract framing that runs from Part II forward. Part VII will make more sense — and feel more actionable — to a reader who has absorbed the preceding six.

What Each Part Covers

Part I — The Philosophy of Boring Code establishes the mindset before any rule is introduced. These four chapters make the argument that boring code is the goal, that clarity is a discipline rather than a default, and that the Zen of Python is a design specification worth taking seriously. Readers who already share this conviction may move through Part I quickly. Those who are skeptical that boring is a meaningful virtue will find the argument here.

Part II — Names Are Contracts is where the technical content begins. Naming is the first interface between code and its readers — every function name either makes an accurate promise or a misleading one. These four chapters cover function names, verb prefix vocabulary, variable names, and the discipline of letting the namespace do its job rather than

repeating it. The verb vocabulary introduced in Chapter 6 and catalogued in Appendix B is used throughout the rest of the book.

Part III — Functions Are Promises covers the design of functions as units of contract. Single responsibility, parameter ordering, keyword-only arguments, and the separation of normalization from processing logic. These chapters are dense with code examples because the principles are most visible in the contrast between a function that is easy to misuse and one that is hard to misuse.

Part IV — Types Are Documentation You Can't Ignore treats type annotations as the machine-readable layer of the contract. The central claim — annotate what you require, not what you happen to be passing — has practical consequences for every function that accepts a collection. The collection type hierarchy introduced in Chapter 13 and referenced in Appendix C applies any time a function accepts more than one item.

Part V — Structure and Control Flow covers the structural choices that communicate intent before a reader processes logic: `match` versus `if`, `pass` versus `...`, specific exception types and exception chaining. These chapters are intentionally shorter than the parts surrounding them. Each addresses a binary distinction — one choice is correct, the other imposes a cost — and the argument does not require extended treatment once the distinction is clear. The common assumption — that `match` is a modern `if`, that `pass` and `...` are interchangeable empty-body placeholders — is wrong in ways that cost every developer who takes them at face value. Part V makes each case precisely and moves on. All three chapters are short enough to read in a single sitting.

Part VI — Documentation as Contract examines the prose of a codebase with the same rigor as its code. These four chapters apply the same contract-first discipline to prose that the earlier parts applied to names and types — treating docstrings as specifications rather than descriptions, and comments as explanations of why rather than what.

Part VII — The Living Standard zooms out from individual decisions to the system that maintains them over time. It introduces the rubric-driven development framework: a way of making code quality standards executable rather than aspirational. A standard that lives only in a document enforces itself only when someone remembers it; a rubric enforces itself every time code is evaluated or reviewed. The framework applies wherever standards need to survive beyond the session in which they were written.

Part VIII — Evidence is a single chapter because the argument it makes is singular. Chapter 28 applies every principle from Parts I through VII to a real codebase, annotating what each decision demonstrates and why it holds. It is not a summary — it is the proof that boring,

applied consistently, produces a codebase that reads exactly as expected.

Chapter Dependencies

Most chapters stand on their own argument. A few carry cross-part dependencies worth knowing if you read out of sequence:

Chapters	Depends on	What it assumes
13	5	The contract/mechanism distinction introduced in Part II
19–22	5	The same contract-first principle applied to prose
24–27	23	The rubric-as-living-document framing from Chapter 23

Part I (Chapters 1–4) is fully standalone but introduces the vocabulary the rest of the book relies on.

How to Use the Appendices

The appendices are reference material, not reading material. They are designed to be consulted at the keyboard, not read end to end.

Appendix A — The Evaluation Checklist condenses each chapter to a single checkable item. Use it during code review, self-evaluation, or AI-assisted evaluation to verify that a file meets the standards the book establishes.

Appendix B — Name Prefix Vocabulary is the full reference for the verb contract vocabulary introduced in Chapter 6. When the right prefix for a new function is not immediately clear, this is the first place to look.

Appendix C — Type Annotation Quick Reference covers the abstract collection type hierarchy with decision heuristics. Use it when choosing between `Iterable`, `Collection`, `Sequence`, and `list` as a parameter annotation.

Appendix D — Docstring Templates provides ready-to-use templates for modules, classes, and functions. It encodes the mood and minimalism rules, and the decision sequence for when to add `Args:`, `Returns:`, and `Raises:` sections.

Appendix E — The Rubric as a Starting Point is the core set of standards from this book formatted for use as an AI session context. Provide it at the start of any AI development session to ensure every evaluation and review applies the same checklist. Appendix E is a starting point: teams will refine it to reflect their own decisions over time.

How This Book Is Different

Python has excellent books about the language itself — how its data model and protocols work, and how to use its more sophisticated features idiomatically. This book is not one of them. It does not go deep into language internals, nor does it compile a list of best practices. Concurrency, async patterns, testing strategy, and package distribution are each their own subject. The argument here is singular: that the highest virtue in professional Python is boringness, and that boringness is not a natural outcome of competence but a discipline that requires deliberate practice.

Part VII is where this book most distinguishes itself. The rubric-driven development framework it introduces — making quality standards executable rather than aspirational, and compounding improvements through a maintained rubric — addresses a problem that language guides and best-practice lists do not: how does a team hold to a consistent standard not just in code review, but across every session and every AI-assisted evaluation? The answer this book offers is the most direct expression of its central argument. Boring standards, maintained consistently, are the team-scale version of boring code.

This book makes the argument in Python, where the language's specific idioms give the principles their precision. The underlying discipline applies in any language where code outlives the session that produced it.

A reader who owns *Effective Python* has a deep guide to using Python's features correctly. A reader who owns *Clean Code* has a language-agnostic case for readable, well-structured software. A reader who owns *A Philosophy of Software Design* has a rigorous argument for reducing complexity. This book does something none of them do: it defines a precise semantic vocabulary for what Python functions promise — through their names, their signatures, their type annotations, and their documentation — and it gives readers the tools to maintain that vocabulary consistently across a codebase over time. None of that is repackaged advice. The verb prefix vocabulary and the named diagnostic tests have no equivalent in the existing literature. Neither does the rubric-driven development framework.

A Note on the Code Examples

Every code example in this book was written to isolate one principle. The before-and-after pairs that appear throughout the chapters are not complete programs — they are the smallest amount of code needed to show the distinction the chapter is making. Comments inside examples explain the principle, not the code itself. In production code, a comment that explained what the code does would be redundant. Here, it serves the reader.

The examples target Python 3.10 and later. Where a feature is version-specific, the chapter notes it. When an example uses a type annotation form that may be unfamiliar — `X | Y` union syntax, `list[str]` rather than `List[str]` — the chapter explains it on first use.

The *Conventions Used in This Book* section that follows this introduction defines the named tests and heuristics that appear throughout the chapters, the docstring discipline, and the reasoning behind its departures from strict style-guide adherence. It also establishes precise meanings for terms like *contract*, *mechanism*, and *observable behavior*.

The book ends with a claim: the highest compliment a reader can pay your code is that it was exactly what they expected. Working through these chapters is practice for writing code that earns that compliment. Not occasionally. Consistently. Across an entire codebase, by a team that maintains a shared rubric and the discipline to apply it.

That is the goal. Boring, done consistently, is the closest thing software development has to art that lasts.

Conventions Used in This Book

This section describes the conventions the book follows so that when you encounter an unusual choice — a docstring style that does not match your linter's expectations, a named test used as a diagnostic tool, a term used in a specific technical sense — you have a reference to consult.

Code Examples

Code examples appear in fenced blocks. When a chapter contrasts two approaches, the comment above each block describes the principle being illustrated:

```
# Wrong — describes the mechanism, not the contract
def find_user_by_json_id(json_payload: str) -> User: ...
```

```
# Correct — describes the contract
def find_user(user_id: int) -> User: ...
```

Comments inside code blocks explain the example. They are not a model for how to comment actual production code — that discipline is covered in Chapter 22.

Named Tests and Heuristics

The book introduces several named diagnostic tests. These are tools you apply at the keyboard, not theoretical principles to memorize. Each test produces a binary answer that guides a specific decision.

The Rewrite Test *(Chapter 19)* Could you rewrite this function's implementation from scratch without changing the docstring? If every word in the docstring were still accurate

after replacing the algorithm, swapping the data store, or switching libraries, the docstring describes the contract. If a rewrite required updating the docstring, the docstring describes the implementation.

The Competent-Caller Test *(Chapter 21)* Would a competent caller already know this? A competent caller is a professional developer who knows Python, understands type annotations, and reads the function name. They are not a beginner who needs every term explained, and not a domain expert who already knows the whole system. If a competent caller would infer the information from the signature and name alone, do not write it in the docstring.

The Rename Test *(Chapter 5)* If you refactored this function's internals tomorrow, would the name still be accurate? A name that requires updating after a refactor is describing the mechanism. A name that would survive any implementation change that preserves the behavior is describing the contract.

The Delete Test *(Chapter 22)* Delete the comment. Read the code without it. Is anything lost? If a reader without the comment would not understand why the code does what it does — and might even "fix" a deliberate decision — the comment earns its place. If nothing is lost, the comment was redundant.

The Enum Heuristic *(Chapter 15)* If you removed the enum from the type signature, would it change how a caller reasons about valid inputs? If yes, the enum is doing real work. If no, the enum is decoration and the values belong as module-level `Final` constants.

The `pass` vs. `...` Test *(Chapter 17)* Ask which statement is true: "Nothing happens here, and that is the right behavior" — or — "The implementation is not here; it lives elsewhere or will be provided by the concrete type." The first answer calls for `pass`. The second calls for `...`.

Docstring Style

The book advocates a docstring discipline that does not strictly follow any single established format. Google Style, NumPy Style, and reStructuredText are widely used and internally consistent, but all tend toward comprehensive documentation by default. This book's approach departs from them in one specific direction: **restraint**.

The standard is **succinct but informative** — not the shortest possible docstring, but the

shortest one that leaves nothing important unsaid. Every sentence must earn its place — anything that remains tells the reader something they could not have known without it.

The principle is the **Minimalism Rule**: use the shortest docstring that completely and accurately describes observable behavior. Start with a single summary sentence. Add sections only when they provide information that the summary cannot.

Mood

The mood of the summary sentence signals what kind of thing is being documented.

What is being documented	Mood	Sounds like
Module	Indicative	"Provides…", "Defines…", "Implements…"
Class	Indicative	"Represents…", "Encapsulates…", "A…"
Function / Method	Imperative	"Return…", "Validate…", "Load…", "Send…"

Function docstrings use imperative mood throughout. "Returns the parsed value." is wrong. "Return the parsed value." is correct. The diagnostic: if the first verb ends in *-s*, drop it.

Bullet Points

Bullet points document non-obvious behavioral details: edge case handling, side effects, ordering guarantees, merge semantics, iteration semantics. Each bullet is a complete sentence — capitalized, with a period. Bullets add information the summary sentence does not cover; they do not restate it.

`Args:`, `Returns:`, and `Raises:` Sections

These sections appear only when they provide information that the summary sentence, the bullet points, the function name, and the type annotations cannot. When the summary already describes what is returned, `Returns:` is redundant. When a parameter's name and type say everything, its `Args:` entry says nothing. When an exception is obvious from the function name, `Raises:` adds no value.

The full `Args:` / `Returns:` / `Raises:` structure is shown in Appendix D as an upper bound, not a target. Most functions need none of it. Some need one section. Very few need all three.

Verb Prefix Vocabulary

Chapter 6 establishes a vocabulary of function-name prefixes and the semantic contracts each one implies. These are not style recommendations — they are signal contracts. Using `get_` on a function that raises tells callers the function is safe to call without error handling, then proves them wrong. Using `is_` on a function with side effects tells callers nothing will change — then the function changes things.

The vocabulary used throughout the book:

Prefix	Contract implied
`get_`	Returns a value; never raises for expected conditions; no side effects
`fetch_`	Retrieves from a remote or external source; may raise on failure
`load_`	Reads from persistent storage; may raise; implies I/O cost
`check_`	Evaluates a condition; may raise; may have side effects
`validate_`	Asserts data meets a specification; raises if not
`ensure_`	Enforces a precondition; raises on failure; returns `None` on success
`normalize_`	Transforms input to canonical form; pure; idempotent; never raises
`parse_`	Converts string or bytes to structured value; raises if invalid
`build_` / `create_`	Constructs and returns a new object (`create_` may have side effects)
`is_` / `has_` / `can_`	Pure boolean predicate; never raises; no side effects

Appendix B provides the full reference table with contract details, edge cases, and the most common points of confusion.

Key Terms

The book uses several terms in a specific technical sense.

Contract. What a function promises to callers: what it accepts, what it returns, what it guarantees, and what it may do to state beyond its return value. The contract is observable — callers can detect it without reading the implementation. The implementation can change in any way that still satisfies the contract.

Mechanism. How a function accomplishes its work: the algorithm, data structure, or library used. The mechanism is an implementation detail. Callers cannot observe it. Names and docstrings that describe the mechanism become lies when the mechanism changes.

Observable behavior. Anything a caller can detect without reading the source: the semantics of return values, side effects, error conditions, and behavioral guarantees. Observable

behavior belongs in contracts. Internal process does not.

Obviously correct. Code that a reader understands immediately, without verification, tracing, or context from surrounding lines. Not "provably correct" or "theoretically sound" — those are different bars. Obviously correct means that the next person to read this code will know what it does, why it does it, and that it is right, without having to reason it through.

Competent caller. A professional developer who knows Python, understands the type annotations, and reads the function name. Not a beginner who needs every term explained, and not a domain expert who already knows the whole system. The imagined reader for whom every docstring is written.

Python Version

Code examples in this book target Python 3.10 and later, which is the minimum version required to run every example as written. The authoritative rubric in `docs/code-evaluation-rubric.md` recommends targeting Python 3.12+ for production code — a higher bar that reflects current best practice. The examples use only what 3.10 provides to illustrate each principle clearly; production code should target 3.12 or later where practical.

Features used in examples include:

- `match` statements (3.10)
- `X | Y` union type syntax in annotations (3.10)
- Built-in generic types in annotations — `list[str]`, `dict[str, int]` — without importing from `typing` (3.9)
- `typing.Protocol` for structural subtyping
- `typing.Final` for typed constants
- `@dataclass(frozen=True)` for immutable data containers

Where an example uses a feature that may be unfamiliar, the chapter explains its purpose in context.

What This Book Does Not Cover

Formatting and style. This book does not cover indentation, line length, import ordering, or whitespace. These concerns are better addressed by automated formatters (Black, Ruff) and import sorters (isort) than by a style guide chapter. The book concerns itself with decisions that tools cannot make.

Testing. Test design, coverage, and test-driven development are outside the scope of this book. The naming and documentation principles apply to test code as fully as to production code, but this book does not address the mechanics of testing.

Performance. The book occasionally acknowledges performance tradeoffs but does not optimize for them. The discipline here is clarity. Where clarity and performance conflict, the book argues for clarity with a documented rationale — which is also good advice about performance optimization.

Project structure, packaging, and deployment. These are real concerns outside this book's scope.

Part I

The Philosophy of Boring Code

Chapter 1 — Coding for Future You

It is 2am. Your phone is on the nightstand and the alert has already fired twice. Production is down.

You open your laptop and follow the stack trace to a function you wrote six months ago. You read the name: `process_data`. You read the body. Nothing is familiar.

The variable `d` appears eleven times. A parameter named `flag` controls a fork in the logic whose meaning you cannot immediately reconstruct. There are two early returns with no comments. The transformation pipeline references three module-level variables that you need to find before you can reason about any of this. You wrote every line of this function. You have no memory of writing any of it.

This is Future You. This is the person your past self left this code for.

The stakes are not always that high. Sometimes it is a side project you return to after a few months away — a game you were building, a script that colorizes terminal output, a utility you wrote for your own use. No alert fired. No system is down. You simply cannot remember what the function does. It made complete sense when you wrote it. The context that made it sensible — the data structure it expected, the edge case it was protecting against, the reason you named the variable the way you did — has faded entirely. The confusion is identical. The pressure is just lower.

Future You is not a hypothetical. Future You is the person who will open this file in three months, or six months, or two years — possibly at 2am, possibly under pressure, possibly on the first week back after time away that has faded the context you had when you wrote it. Future You has no access to the meeting where this design was decided. There is no colleague to ask about the flag. Future You has only what is written.

This creates an asymmetry that every programmer experiences but few consciously design

around. Present You is rich with context. You know why `flag` is `True` by default. You know what `d` represents because you were just looking at the data model. You know the early returns are safe because you verified the invariants last Tuesday. None of that knowledge is in the code. All of it lives in your head, where it will not survive the next six months.

Writing code for Present You is the natural mode. It is fast. Names are short because you know what they mean. The comments are sparse because the logic is obvious right now. The function does two things because you needed both things, and it was faster to combine them. This is not laziness. It is a rational response to the context you have. And it does not just feel efficient — it feels like craft. The code is tight, the logic is clear in your head, and the solution is elegant. None of that is wrong. The problem is that the context is temporary and the code is not.

Writing code for Future You means accepting that you are not the only reader. In fact, you are not even the most important reader. The most important reader is the person who will work with this code after the context has faded — which is to say, you, six months from now. Or a colleague who joined the project last week. Or anyone who needs to change this function without breaking something else.

Think of it this way: you are writing a letter to a stranger who happens to share your name. Everything that makes the code clear or opaque, easy or hard to modify, self-explanatory or riddled with questions — all of it is language in that letter. Every function name is a sentence. Every comment is a clarification. A confusing structure is a paragraph that trails off into illegibility. The stranger who will read it has nothing but what you wrote.

Here is what that looks like in practice. Consider two versions of the same function — one written for Present You, one written for Future You.

```
def process(d, flag=False):
    r = db.query(d)
    if flag:
        r = normalize(r)
    return r
```

And:

```
def load_user_record(user_id: int, *, apply_normalization: bool = False) -> UserRecord:
    """Load a user record from the database by ID.

    - If apply_normalization is True, applies the standard field normalization
      pipeline before the record is returned.
```

```
    """
    record = db.query(user_id)
    if apply_normalization:
        record = normalize(record)
    return record
```

Both functions do the same thing. One was written for Present You. The other was written for Future You. The first took less time to write. The second will take less time to read — every time it is read, by everyone who ever reads it, for as long as it exists.

The math is not complicated. If a badly named function takes thirty seconds longer to understand, ten people reading it costs five minutes. A hundred people reading it costs nearly an hour. Those seconds you saved writing `d` instead of `user_id` — someone else is paying them back, every time. You wrote the check. They pay it.

This is the operating principle behind everything in this book. Not rules for their own sake. Not compliance with a style guide. The distinction between code that is a gift and code that is a burden.

Gift code is clear about what it does — names that make promises and keep them, structure that reflects logic, documentation that describes what a caller needs to know rather than what the author needed to remember. When Future You opens a function written as a gift, the experience is nearly invisible — you read it, understand it, make the change, and move on. The function gets out of the way.

Burden code makes you work. The names are opaque or misleading. Convenience shaped the structure, not clarity. The documentation, if any exists, describes the implementation rather than the contract. When Future You opens a function written as a burden, the first task is not to solve the problem — it is to decode the function well enough to approach the problem. And you are never quite sure you decoded it correctly.

Present You always knows which one you are writing. You know when you are naming something `d` because you are in a hurry. You know when you are skipping the docstring because the function seems obvious right now. And you know when the function is doing two things, and you combined them anyway. The question is whether you are willing to treat Future You as a collaborator worth the extra three minutes.

The chapters that follow are organized around the decisions that accumulate into either gift

code or burden code: names, function signatures, type annotations, structure, documentation, and the systems that maintain standards over time. Each is a specific place where Present You makes a choice that Future You will live with. None of the principles are difficult. Most of them take longer to explain than to apply. But applying them consistently — across every file, every function, every session — is what separates code that lasts from code that merely survives.

Future You is reading everything you write. Write accordingly.

Part II

Names Are Contracts

Chapter 5 — Name the Thing, Not the Mechanism

The next time you refactor a function and feel the urge to rename it, stop. That urge is a diagnostic. It means the name you gave it described how the function worked, not what it did — and now that the internals have changed, the name has become a lie. Functions named for their mechanisms do not survive refactoring. Functions named for their contracts do.

This is the first and most fundamental rule of naming: name what a function returns or accomplishes, not how it accomplishes it.

The distinction is between contract and implementation. A function's contract is what it promises to callers: what it accepts, what it returns, what it guarantees. A function's implementation is how it delivers on that promise: the libraries it uses, the data format it parses, the storage system it queries. The contract belongs in the name. The implementation belongs in the body.

Consider a function that reads a configuration file:

```
def parse_yaml_config(path: Path) -> Config:
    with open(path) as f:
        data = yaml.safe_load(f)
    return Config(**data)
```

The name `parse_yaml_config` describes two implementation details: the file format (YAML) and the operation (parsing). Now consider what happens when the team decides to migrate to TOML:

```
def parse_yaml_config(path: Path) -> Config:
    with open(path) as f:
        data = tomllib.load(f)
```

```
    return Config(**data)
```

The function now parses TOML. The name says YAML. Every call site is a lie, and fixing it means a refactor that touches the name everywhere it appears — not because the contract changed, but because the name was never describing the contract.

```
def load_config(path: Path) -> Config:
    with open(path) as f:
        data = tomllib.load(f)
    return Config(**data)
```

`load_config` survives. The format is an implementation detail. What the function does — load configuration — did not change when the format changed. Callers who wrote `config = load_config(path)` do not care whether the file is YAML or TOML. They asked for configuration. They received configuration. The name remained honest.

This pattern appears wherever serialization formats, storage mechanisms, or parsing libraries make it into function names:

```
# Mechanism names — tied to implementation
def parse_json_user(payload: bytes) -> User: ...


def read_csv_report(path: Path) -> Report: ...


def deserialize_msgpack_event(data: bytes) -> Event: ...


def fetch_user_from_database(user_id: int) -> User: ...
```

```
# Contract names — tied to what is returned
def load_user(payload: bytes) -> User: ...


def load_report(path: Path) -> Report: ...


def load_event(data: bytes) -> Event: ...


def get_user(user_id: int) -> User: ...
```

The mechanism names are not wrong about what the functions do today. They are wrong about what the names are for. Names are not documentation of the current implementation. Names are the interface between the function and everything that calls it — and that interface should survive the implementation changing.

A name that includes the storage layer (`_from_database`) is particularly fragile. Storage layers are exactly the kind of implementation detail that changes: a cache gets added, a database gets swapped, a remote API gets replaced by a local store. `fetch_user_from_database` becomes wrong the moment a Redis cache sits in front of the database. `get_user` does not — it describes what the caller receives, not where it came from.

There is a test for this. Apply it to any function name you are uncertain about:

If you refactored this function's internals tomorrow, would the name still be accurate?

If yes, the name describes the contract. If no, the name describes the implementation.

Apply the test to `parse_yaml_config`: refactor the internals to TOML. Would you rename it? Yes. The name fails the test. Apply it to `load_config`: refactor the internals to TOML. Would you rename it? No. The contract is unchanged. The name passes.

The rename urge is not just a hint about the current name. It is a diagnostic for a category of technical debt: every function whose name is coupled to its implementation will require renaming when the implementation changes. In a large codebase, those renames are expensive — not because the operation is hard, but because every call site has to be found and updated, documentation has to be revised, and the history of the change has to be explained. All of that cost traces back to a name that described how instead of what.

The same principle applies to test function names. A test named for the mechanism it exercises will need renaming when the implementation changes — even if the behavior being verified has not changed at all.

```
# Mechanism names — describe how the production code works, not what it promises
def test_parses_yaml_using_safe_load(): ...


def test_queries_database_for_active_users(): ...


def test_regex_rejects_email_without_at_sign(): ...
```

```python
# Contract names — describe the observable behavior being verified
def test_load_config_returns_defaults_when_field_is_absent(): ...


def test_find_active_users_excludes_suspended_accounts(): ...


def test_invalid_email_address_rejected(): ...
```

When `load_config` migrates from YAML to TOML, the first version of the config test breaks by name even though nothing about the behavior being verified has changed. The second version survives — it names what the function promises to return, not the path it takes to get there.

Apply the Rename Test to test functions the same way you apply it to production functions: if the implementation changes while the contract holds, the test name should not need to change. A test name that describes the mechanism fails the test. A test name that describes the observable behavior passes it.

There is a corollary worth stating explicitly. Sometimes a refactor genuinely changes a function's contract — and in those cases, renaming is correct.

```python
# Before: loads from a local file
def load_user(user_id: int) -> User: ...


# After: now makes a network call to a remote API
def fetch_user(user_id: int) -> User: ...
```

This rename is right. The contract changed: the function now performs I/O with latency and failure modes that a local file load does not have. Callers who wrote `user = load_user(id)` without a `try/except` were safe before. They may not be safe now. The rename signals that something meaningful has changed.

The distinction is between renaming because the *implementation* changed (a symptom of a mechanism name) and renaming because the *contract* changed (correct response to a real semantic shift). Mechanism names force the first kind of rename. Contract names limit it to the second.

Naming the thing rather than the mechanism is not always straightforward. Sometimes the mechanism is so central to the contract that they cannot be cleanly separated. A function named `parse_iso8601_timestamp` describes both a mechanism (ISO 8601 parsing) and a contract (returns a timestamp). If the format is fixed by an external system and will never change, the format in the name may be warranted — it is part of the contract, not an incidental detail of the implementation.

The question to ask is: would a caller who does not know how this function works need to know the mechanism to use it correctly? If the caller needs to supply ISO 8601-formatted input, the format is part of the contract and belongs in the name or the documentation. If the caller just needs a `User` object and the format is irrelevant to them, the format belongs only in the implementation.

When in doubt, name the outcome. The outcome is what callers receive; the mechanism is what they should not have to think about.

The principle carries forward into every naming decision in Part II. Variables that describe their role rather than their type, namespaces that add information rather than repeat context, verbs that signal contracts rather than operations — all of these are expressions of the same idea. Names face outward, toward the reader. They describe what a thing is for, not how it was made. The implementation is private. The contract is public. Name accordingly.

About the Author

Roth Earl has been writing programs since the moment he discovered that JavaScript could make a plane fly across a screen — and that clicking on it could make the plane explode. The game was simple, the sounds were questionable, and the animation was held together by determination and the kind of optimism that only comes from not yet knowing what you don't know. He has been chasing that feeling ever since.

He began his career in 2005 as a software developer and transitioned to test engineering roughly twelve years ago — a move that changed how he reads code. Where a developer reads code to extend it, a test engineer reads code to understand what it actually promises: what the function will do, what it can fail on, and whether its name is telling the truth. That perspective, applied across two decades and a wide range of codebases, is the foundation of this book.

He has no prior books or formal academic credentials in this area. His authority is practical: twenty years of writing code, reading code, inheriting code, and paying the maintenance costs of decisions that made sense at the time. The argument in this book was not constructed from theory. It was accumulated from the experience of opening files that should have taken thirty seconds to understand and taking twenty minutes instead — and from deciding, at some point, to write code that does not do that to the next person.

The book you just finished came from a conviction that arrived, as most good ones do, through experience: in the age of AI-assisted development, code that is painfully obvious is the only kind worth writing. Not because it is the easiest to produce — it is not — but because it is the only kind that Future You, Future Roth, and every developer who encounters it will understand immediately — even, he notes with some affection, as memory starts to do what memory eventually does to all of us.